30 | 部门响应：设备如何处理内核I/O包？

2021-07-16 LMOS

《操作系统实战45讲》 课程介绍 >

讲述：陈晨

时长 15:14 大小 13.97M

你好，我是 LMOS。

在上一课中，我们实现了建立设备的接口，这相当于制定了部门的相关法规，只要遵守这些法规就能建立一个部门。当然，建立了一个部门，是为了干活的，吃空饷可不行。

其实一个部门的职责不难确定，它应该能对上级下发的任务作出响应，并完成相关工作，而这对应到设备，就是如何处理内核的 I/O 包，这节课我们就来解决这个问题。

首先，我们需要搞清楚什么是 I/O 包，然后实现内核向设备发送 I/O 包的工作。最后，还会带你一起来完成一个驱动实例，用于处理 I/O 包，这样你就能真正理解这里的来龙去脉了。

好，让我们开始今天的学习吧！代码你可以从这里下载。

什么是 I/O 包

就像你要给部门下达任务时，需要准备材料报表之类的东西。同样，内核要求设备做什么事情，完成什么功能，必须要告诉设备的驱动程序。

内核要求设备完成任务，无非是调用设备的驱动程序函数，把完成任务的细节用参数的形式传递给设备的驱动程序。

由于参数很多，而且各种操作所需的参数又不相同，所以我们就想到了更高效的管理方法，也就是把各种操作所需的各种参数封装在一个数据结构中，称为 I/O 包，这样就可以统一驱动程序功能函数的形式了。

思路理清以后，现在我们来设计这个数据结构，如下所示。

复制代码

```
typedef struct s_OBJNODE
{
    spinlock_t  on_lock;        //自旋锁
    list_h_t    on_list;        //链表
    sem_t       on_complesem;   //完成信号量
    uint_t      on_flgs;        //标志
    uint_t      on_stus;        //状态
    sint_t      on_opercode;    //操作码
    uint_t      on_objtype;     //对象类型
    void*       on_objadr;      //对象地址
    uint_t      on_acsflgs;     //访问设备、文件标志
    uint_t      on_acsstus;     //访问设备、文件状态
    uint_t      on_currops;     //对应于读写数据的当前位置
    uint_t      on_len;         //对应于读写数据的长度
    uint_t      on_ioctrd;      //IO控制码
    buf_t       on_buf;         //对应于读写数据的缓冲区
    uint_t      on_bufcurops;   //对应于读写数据的缓冲区的当前位置
    size_t      on_bufsz;       //对应于读写数据的缓冲区的大小
    uint_t      on_count;       //对应于对象节点的计数
    void*       on_safedsc;     //对应于对象节点的安全描述符
    void*       on_fname;       //对应于访问数据文件的名称
    void*       on_finode;      //对应于访问数据文件的结点
    void*       on_extp;        //用于扩展
}objnode_t;
```

现在你可能还无法从 objnode_t 这个名字看出它跟 I/O 包的关系。但你从刚才的代码里可以看出，objnode_t 的数据结构中包括了各个驱动程序功能函数的所有参数。

等我们后面讲到 API 接口时，你会发现，objnode_t 结构不单是完成了 I/O 包传递参数的功能，它在整个 I/O 生命周期中，都起着重要的作用。这里为了好理解，我们就暂且把 objnode_t 结构当作 I/O 包来看。

创建和删除 I/O 包

刚才，我们已经定义了 I/O 包也就是 objnode_t 结构，但若是要使用它，就必须先把它建立好。

根据以往的经验，你应该已经猜到了，这里创建 I/O 包就是在内存中建立 objnode_t 结构的实例变量并初始化它。由于这是一个全新的模块，所以我们要先在 cosmos/kernel/ 目录下建立一个新的 krlobjnode.c 文件，在这个文件中写代码，如下所示。

复制代码

```
//建立objnode_t结构
objnode_t *krlnew_objnode()
{
    objnode_t *ondp = (objnode_t *)krlnew((size_t)sizeof(objnode_t));//分配objr
    if (ondp == NULL)
    {
        return NULL;
    }
    objnode_t_init(ondp);//初始化objnode_t结构
    return ondp;
}
//删除objnode_t结构
bool_t krldel_objnode(objnode_t *onodep)
{
    if (krldelete((adr_t)onodep, (size_t)sizeof(objnode_t)) == FALSE)//删除objr
    {
        hal_sysdie("krldel_objnode err");
        return FALSE;
    }
    return TRUE;
}
```

上述代码非常简单，主要完成了建立、删除 objnode_t 结构这两件事，其实说白了就是分配和释放 objnode_t 结构的内存空间。

这里再一次体现了**内存管理组件在操作系统内核之中的重要性**，objnode_t_init 函数会初始化 objnode_t 结构中的字段，因为其中有自旋锁、链表、信号量，而这些结构并不能简单地初始为 0，否则可以直接使用 memset 之类的函数把那个内存空间清零就行了。

向设备发送 I/O 包

现在我们假定在上层接口函数中，已经建立了一个 I/O 包（即 objnode_t 结构），并且把操作码、操作对象和相关的参数信息填写到了 objnode_t 结构之中。那么下一步，就需要把这个 I/O 发送给具体设备的驱动程序，以便驱动程序完成具体工作。

我们需要定义实现一个函数，专门用于完成这个功能，它标志着一个设备驱动程序开始运行，经它之后内核就实际的控制权交给驱动程序，由驱动程序代表内核操控设备。

下面，我们就来写好这个函数，不过这个函数属于驱动模型函数，所以要在 krldevice.c 文件中实现这个函数。代码如下所示。

复制代码

```
//发送设备IO
drvstus_t krldev_io(objnode_t *nodep)
{
    //获取设备对象
    device_t *devp = (device_t *)(nodep->on_objadr);
    if ((nodep->on_objtype != OBJN_TY_DEV && nodep->on_objtype != OBJN_TY_FIL)
    {//检查操作对象类型是不是文件或者设备，对象地址是不是为空
        return DFCERRSTUS;
    }
    if (nodep->on_opercode < 0 || nodep->on_opercode >= IOIF_CODE_MAX)
    {//检查IO操作码是不是合乎要求
        return DFCERRSTUS;
    }
    return krldev_call_driver(devp, nodep->on_opercode, 0, 0, NULL, nodep);//调
}
//调用设备驱动
drvstus_t krldev_call_driver(device_t *devp, uint_t iocode, uint_t val1, uint_
{
    driver_t *drvp = NULL;
    if (devp == NULL || iocode >= IOIF_CODE_MAX)
    {//检查设备和IO操作码
        return DFCERRSTUS;
    }
    drvp = devp->dev_drv;
    if (drvp == NULL)//检查设备是否有驱动程序
    {
        return DFCERRSTUS;
```

```
    }
    //用IO操作码为索引调用驱动程序功能分派函数数组中的函数
    return drvp->drv_dipfun[iocode](devp, p2);
}
```

krldev_io 函数，只接受一个参数，也就是 objnode_t 结构的指针。它会首先检查 objnode_t 结构中的 IO 操作码是不是合乎要求的，还要检查被操作的对象即设备是不是为空，然后调用 krldev_call_driver 函数。

这个 krldev_call_driver 函数会再次确认传递进来的设备和 IO 操作码，然后重点检查设备有没有驱动程序。这一切检查通过之后，我们就用 IO 操作码为索引调用驱动程序功能分派函数数组中的函数，并把设备和 objnode_t 结构传递进去。有没有觉得眼熟？没错，这正是我们前面课程中对驱动程序的设计。

好了，现在一个设备的驱动程序就能正式开始工作，开始响应处理内核发来的 I/O 包了。可是我们还没有驱动呢，所以下面我们就去实现一个驱动程序。

驱动程序实例

现在我们一起来实现一个真实而且简单的设备驱动程序，就是 systick 设备驱动，它是我们 Cosmos 系统的心跳，systick 设备的主要功能和作用是每隔 1ms 产生一个中断，相当于一个定时器，每次时间到达就产生一个中断向系统报告又过了 1ms，相当于千分之一秒，即每秒钟内产生 1000 次中断。

对于现代 CPU 的速度来说，这个中断频率不算太快。x86 平台上有没有这样的定时器呢？当然有，其中 8254 就是一个古老且常用的定时器，对它进行编程设定，它就可以周期的产生定时器中断。

这里我们就以 8254 定时器为基础，实现 Cosmos 系统的 systick 设备。我们先从 systick 设备驱动程序的整体框架入手，然后建立 systick 设备，最后一步一步实现 systick 设备驱动程序。

systick 设备驱动程序的整体框架

在前面的课程中，我们已经了解了在 Cosmos 系统下，一个设备驱动程序的基本框架，但是我们没有深入具体化。

所以，这里我会带你从全局好好了解一个真实的设备，它的驱动程序应该至少有哪些函数。由于这是个驱动程序，我们需要在 cosmos/drivers/ 目录下建立一个 drvtick.c 文件，在 drvtick.c 文件中写入以下代码，如下所示。

复制代码

```
//驱动程序入口和退出函数
drvstus_t systick_entry(driver_t *drvp, uint_t val, void *p)
{
    return DFCERRSTUS;
}
drvstus_t systick_exit(driver_t *drvp, uint_t val, void *p)
{
    return DFCERRSTUS;
}
//设备中断处理函数
drvstus_t systick_handle(uint_t ift_nr, void *devp, void *sframe)
{
    return DFCEERSTUS;
}
//打开、关闭设备函数
drvstus_t systick_open(device_t *devp, void *iopack)
{
    return DFCERRSTUS;
}
drvstus_t systick_close(device_t *devp, void *iopack)
{
    return DFCERRSTUS;
}
//读、写设备数据函数
drvstus_t systick_read(device_t *devp, void *iopack)
{
    return DFCERRSTUS;
}
drvstus_t systick_write(device_t *devp, void *iopack)
{
    return DFCERRSTUS;
}
//调整读写设备数据位置函数
drvstus_t systick_lseek(device_t *devp, void *iopack)
{
    return DFCERRSTUS;
}
//控制设备函数
drvstus_t systick_ioctrl(device_t *devp, void *iopack)
{
    return DFCERRSTUS;
}
//开启、停止设备函数
drvstus_t systick_dev_start(device_t *devp, void *iopack)
```

```
{
    return DFCERRSTUS;
}
drvstus_t systick_dev_stop(device_t *devp, void *iopack)
{
    return DFCERRSTUS;
}
//设置设备电源函数
drvstus_t systick_set_powerstus(device_t *devp, void *iopack)
{
    return DFCERRSTUS;
}
//枚举设备函数
drvstus_t systick_enum_dev(device_t *devp, void *iopack)
{
    return DFCERRSTUS;
}
//刷新设备缓存函数
drvstus_t systick_flush(device_t *devp, void *iopack)
{
    return DFCERRSTUS;
}
//设备关机函数
drvstus_t systick_shutdown(device_t *devp, void *iopack)
{
    return DFCERRSTUS;
}
```

以上就是一个驱动程序必不可少的函数，**在各个函数可以返回一个错误状态，而不做任何实际工作，但是必须要有这个函数。**这样在内核发来任何设备功能请求时，驱动程序才能给予适当的响应。这样，一个驱动程序的整体框架就确定了。

写好了驱动程序的整体框架，我们这个驱动就完成了一半。下面我们来一步一步来实现它。

systick 设备驱动程序的入口

我们先来写好 systick 设备驱动程序的入口函数。那这个函数用来做什么呢？其实我们在上一节课就详细讨论过，无非是建立设备，向内核注册设备，安装中断回调函数等操作，所以这里不再赘述。

我们直接写出这个函数，如下所示。

复制代码

```
drvstus_t systick_entry(driver_t* drvp,uint_t val,void* p)
{
    if(drvp==NULL) //drvp是内核传递进来的参数，不能为NULL
    {
        return DFCERRSTUS;
    }
    device_t* devp=new_device_dsc();//建立设备描述符结构的变量实例
    if(devp==NULL)//不能失败
    {
        return DFCERRSTUS;
    }
    systick_set_driver(drvp);
    systick_set_device(devp,drvp);//驱动程序的功能函数设置到driver_t结构中的drv_dip
    if(krldev_add_driver(devp,drvp)==DFCERRSTUS)//将设备挂载到驱动中
    {
        if(del_device_dsc(devp)==DFCERRSTUS)//注意释放资源。
        {
            return DFCERRSTUS;
        }
        return DFCERRSTUS;
    }
    if(krlnew_device(devp)==DFCERRSTUS)//向内核注册设备
    {
        if(del_device_dsc(devp)==DFCERRSTUS)//注意释放资源
        {
            return DFCERRSTUS;
        }
        return DFCERRSTUS;
    }
    //安装中断回调函数systick_handle
    if(krlnew_devhandle(devp,systick_handle,20)==DFCERRSTUS)
    {
        return DFCERRSTUS;  //注意释放资源。
    }
    init_8254();//初始化物理设备
    if(krlenable_intline(0x20)==DFCERRSTUS)
    {
        return DFCERRSTUS;
    }
    return DFCOKSTUS;
}
```

你可能非常熟悉这部分代码，没错，这正是上节课中，我们的那个驱动程序入口函数的实例。

不过在上节课里，我们主要是要展示一个驱动程序入口函数的流程。这里却是要投入工作的真实设备驱动。

最后的 **krlenable_intline 函数**，它的主要功能是开启一个中断源上的中断。而 init_8254 函数则是为了初始化 8254，它就是一个古老且常用的定时器。这两个函数非常简单，我已经帮写好了。

但是这样还不够，有了驱动程序入口函数，驱动程序并不会自动运行。根据前面我们的设计，需要把这个驱动程序入口函数放入驱动表中。

下面我们就把这个 systick_entry 函数，放到驱动表里，代码如下所示。

复制代码

```
//cosmos/kernel/krlglobal.c
KRL_DEFGLOB_VARIABLE(drventyexit_t,osdrvetytabl)[]={systick_entry,NULL};
```

有了刚才这步操作之后，Cosmos 在启动的时候，就会执行初始驱动初始化 init_krldriver 函数，接着这个函数就会启动运行 systick 设备驱动程序入口函数。我们的 systick_entry 函数一旦执行，就会建立 systick 设备，不断的产生时钟中断。

配置设备和驱动

在驱动程序入口函数中，除了那些标准的流程之外，我们还要对设备和驱动进行适当的配置，就是设置一些标志、状态、名称、驱动功能派发函数等等。有了这些信息，设备才能加入到驱动程序中，然后注册到内核，这样才能被内核所识别。

好，让我们先来实现设置驱动程序的函数，它主要设置设备驱动程序的名称、功能派发函数，代码如下。

复制代码

```
void systick_set_driver(driver_t *drvp)
{
    //设置驱动程序功能派发函数
    drvp->drv_dipfun[IOIF_CODE_OPEN] = systick_open;
    drvp->drv_dipfun[IOIF_CODE_CLOSE] = systick_close;
    drvp->drv_dipfun[IOIF_CODE_READ] = systick_read;
    drvp->drv_dipfun[IOIF_CODE_WRITE] = systick_write;
    drvp->drv_dipfun[IOIF_CODE_LSEEK] = systick_lseek;
    drvp->drv_dipfun[IOIF_CODE_IOCTRL] = systick_ioctrl;
    drvp->drv_dipfun[IOIF_CODE_DEV_START] = systick_dev_start;
    drvp->drv_dipfun[IOIF_CODE_DEV_STOP] = systick_dev_stop;
    drvp->drv_dipfun[IOIF_CODE_SET_POWERSTUS] = systick_set_powerstus;
```

```
    drvp->drv_dipfun[IOIF_CODE_ENUM_DEV] = systick_enum_dev;
    drvp->drv_dipfun[IOIF_CODE_FLUSH] = systick_flush;
    drvp->drv_dipfun[IOIF_CODE_SHUTDOWN] = systick_shutdown;
    drvp->drv_name = "systick0drv";//设置驱动程序名称
    return;
}
```

上述代码的功能并不复杂，我一说你就能领会。systick_set_driver 函数，无非就是将 12 个驱动功能函数的地址，分别设置到 driver_t 结构的 drv_dipfun 数组中。其中，驱动功能函数在该数组中的元素位置，正好与 IO 操作码一一对应，当内核用 IO 操作码调用驱动时，就是调用了这个数据中的函数。最后，我们将驱动程序的名称设置为 systick0drv。

新建的设备也需要配置相关的信息才能工作，比如需要指定设备，设备状态与标志，设备类型、设备名称这些信息。尤其要注意的是，设备类型非常重要，内核正是通过类型来区分各种设备的，下面我们写个函数，完成这些功能，代码如下所示。

复制代码

```
void systick_set_device(device_t *devp, driver_t *drvp)
{
    devp->dev_flgs = DEVFLG_SHARE;//设备可共享访问
    devp->dev_stus = DEVSTS_NORML;//设备正常状态
    devp->dev_id.dev_mtype = SYSTICK_DEVICE;//设备主类型
    devp->dev_id.dev_stype = 0;//设备子类型
    devp->dev_id.dev_nr = 0; //设备号
    devp->dev_name = "systick0";//设置设备名称
    return;
}
```

上述代码中，systick_set_device 函数需要两个参数，但是第二个参数暂时没起作用，而第一个参数其实是一个 device_t 结构的指针，在 systick_entry 函数中调用 new_device_dsc 函数的时候，就会返回这个指针。后面我们会把设备加载到内核中，那时这个指针指向的设备才会被注册。

打开与关闭设备

其实对于 systick 这样设备，主要功能是定时中断，还不能支持读、写、控制、刷新、电源相关的功能，就算内核对 systick 设备发起了这样的 I/O 包，systick 设备驱动程序相关的功能函数也只能返回一个错误码，表示不支持这样的功能请求。

但是，打开与关闭设备这样的功能还是应该要实现。下面我们就来实现这两个功能请求函数，代码如下所示。

复制代码

```
//打开设备
drvstus_t systick_open(device_t *devp, void *iopack)
{
    krldev_inc_devcount(devp);//增加设备计数
    return DFCOKSTUS;//返回成功完成的状态
}
//关闭设备
drvstus_t systick_close(device_t *devp, void *iopack)
{
    krldev_dec_devcount(devp);//减少设备计数
    return DFCOKSTUS;//返回成功完成的状态
}
```

这样，打开与关闭设备的功能就实现了，只是简单地增加与减少设备的引用计数，然后返回成功完成的状态就行了。而增加与减少设备的引用计数，是为了统计有多少个进程打开了这个设备，当设备引用计数为 0 时，就说明没有进程使用该设备。

systick 设备中断回调函数

对于 systick 设备来说，重要的并不是打开、关闭，读写等操作，而是 systick 设备产生的中断，以及在中断回调函数中执行的操作，即周期性的执行系统中的某些动作，比如更新系统时间，比如控制一个进程占用 CPU 的运行时间等，这些操作都需要在 systick 设备中断回调函数中执行。

按照前面的设计，systick 设备每秒钟产生 1000 次中断，那么 1 秒钟就会调用 1000 次这个中断回调函数，这里我们只要写出这个函数就行了，因为安装中断回调函数的思路，我们在前面的课程中已经说过了（可以回顾上节课），现在我们直接实现这个中断函数，代码可以像后面这样写。

复制代码

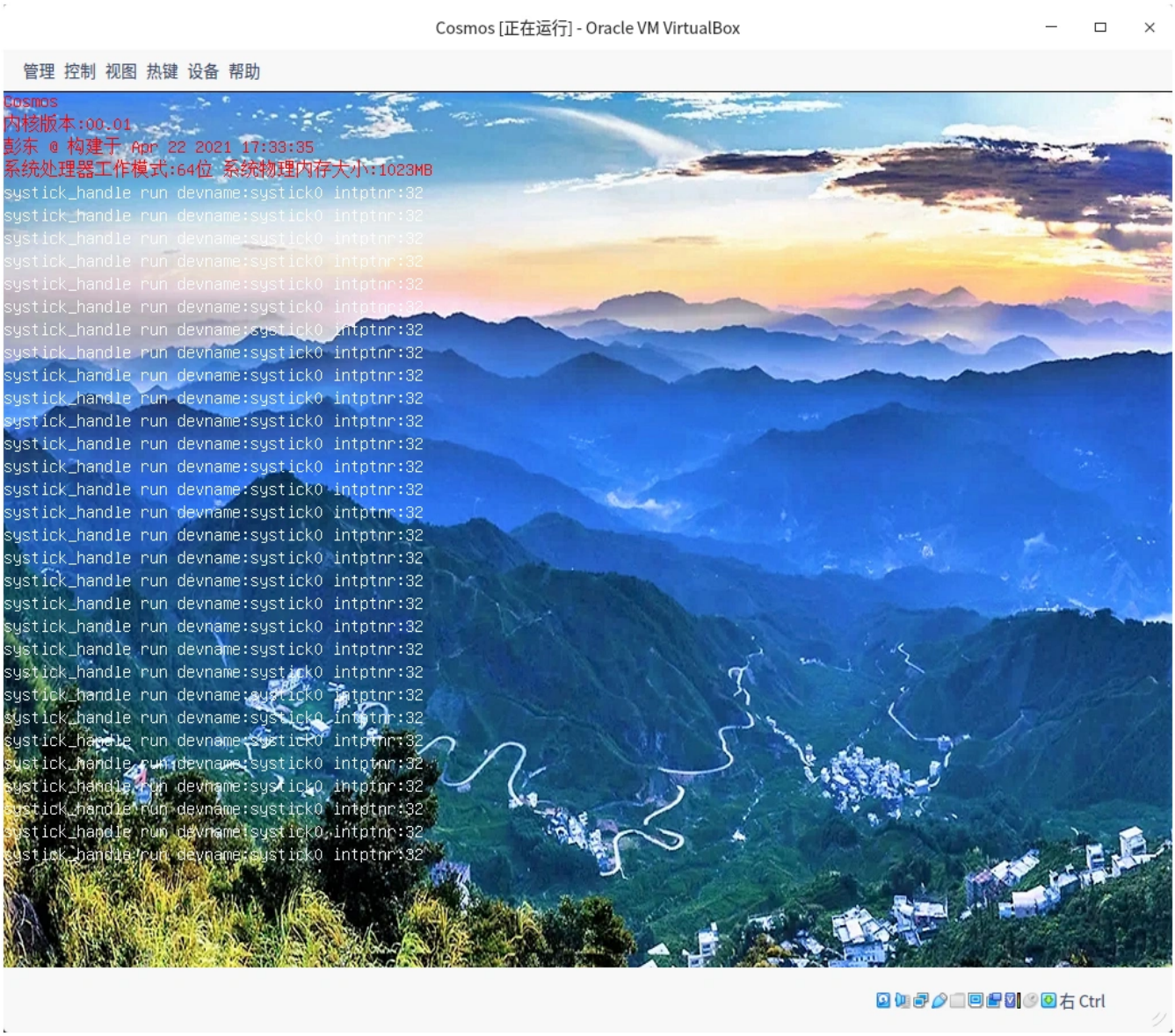
```
drvstus_t systick_handle(uint_t ift_nr, void *devp, void *sframe)
{
    kprint("systick_handle run devname:%s intptnr:%d\n", ((device_t *)devp)->d
    return DFCOKSTUS;
}
```

这个中断回调函数，暂时什么也没干，就输出一条信息，让我们知道它运行了，为了直观观察它运行了，我们要对内核层初始化函数修改一下，禁止进程运行，以免进程输出的信息打扰我们观察结果，修改的代码如下所示。

复制代码

```
void init_krl()
{
    init_krlmm();
    init_krldevice();//初始化设备
    init_krldriver();//初始化驱动程序
    init_krlsched();
    //init_krlcpuidle();禁止进程运行
    STI();//打开CPU响应中断的能力
    die(0);//进入死循环
    return;
}
```

下面，我们打开终端切到 Cosmos 目录下，执行 make vboxtest 指令，如果不出意外，我们将会中看到如下界面。



测试中断回调函数

上图中的信息，会不断地滚动出现，信息中包含设备名称和中断号，这标志着我们中断回调函数的运行正确无误。

当然，如果我们费了这么功夫搞了中断回调函数，就只是为了输出信息，那也太不划算了，我们当然有更重要的事情要做，你还记得之前讲过的进程知识吗？这里我再帮你理一理思路。

我们在每个进程中都要主动调用进程调度器函数，否则进程就会永远霸占 CPU，永远运行下去。这是因为，我们没有定时器可以周期性地检查进程运行了多长时间，如果进程的运行时间超过了，就应该强制调度，让别的进程开始运行。

更新进程运行时间的代码，我已经帮你写好了，你只需要在这个中断回调函数中调用就好了，代码如下所示。

复制代码

```
drvstus_t systick_handle(uint_t ift_nr, void *devp, void *sframe)
{
    krlthd_inc_tick(krlsched_retn_currthread());//更新当前进程的tick
    return DFCOKSTUS;
}
```

这里的 krlthd_inc_tick 函数需要一个进程指针的参数，而 krlsched_retn_currthread 函数是返回当前正在运行进程的指针。在 krlthd_inc_tick 函数中对进程的 tick 值加 1，如果大于 20（也就是 20 毫秒）就重新置 0，并进行调度。

下面，我们把内核层初始化函数恢复到原样，重新打开终端切到 cosmos 目录下，执行 make vboxtest 指令，我们就将会看到如下界面。

测试进程运行时间更新

我们可以看到，进程 A、进程 B，还有调度器交替输出的信息。这已经证明我们更新进程运行时间，检查其时间是否用完并进行调度的代码逻辑，都是完全正确的，恭喜你走到了这一步！

至此，我们的 systick 驱动程序就实现了，它非常简单，但却包含了一个驱动程序完整实现。同时，这个过程也一步步验证了我们对驱动模型的设计是正确的。

重点回顾

又到课程的结尾，到此为止，我们了解了实现一个驱动程序完整过程，虽然我们只是驱动了一个定时器设备，使之周期性的产生定时中断。在定时器设备的中断回调函数中，我们调用了更新进程时间的函数，达到了这样的目的：在进程运行超时的情况下，内核有能力夺回 CPU，调度别的进程运行。

现在我来为你梳理一下重点。

1. 为了搞清楚设备如何处理 I/O 包，我们了解了什么 I/O 包，写好了处理建立、删除 I/O 包的代码。

2. 要使设备完成相应的功能，内核就必须向设备驱动发送相应的 I/O 包，在 I/O 包提供相应 IO 操作码和适当的参数。所以，我们动手实现了向设备发送 I/O 包并调用设备驱动程序的机制。

3. 一切准备就绪之后，我们建立了 systick 驱动程序实例，这是一个完整的驱动程序，它支持打开关闭和周期性产生中断的功能请求。通过这个实例，让我们了解了一个真实设备驱动的实现以及它处理内核 I/O 包的过程。

你可能对这样简单的驱动程序不够满意，也不能肯定我们的驱动模型是不是能适应大多数场景，请不要着急，在后面讲到文件系统时，我们会实现一个更为复杂的驱动程序。

思考题

请你想一想，为什么没有 systick 设备这样周期性的产生中断，进程就有可能霸占 CPU 呢？

欢迎你在留言区跟我交流互动，也欢迎你把这节课分享给身边的同事、朋友，一起实践驱动程序的实例。

好，我是 LMOS，我们下节课见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 29 | 部门建立：如何在内核中注册设备？

下一篇 31 | 瞧一瞧Linux：如何获取所有设备信息？

精选留言 (4)

写留言

罗乾林
2021-07-16

进程调度发生在用户态和内核态切换，如果一个应用程序只在用户态做计算不进行系统调用，系统将没有机会完成进程调度。定时器的出现就能打破这一问题，使系统进入中断处理程序，进而完成进程调度

作者回复: 正确 老铁

pedro
2021-07-16

因为之前实现的调度主要看优先级和运行时间，如果没有 systick 中断来记录运行时间，那么调度器就无法根据运行时间来强制休眠高优先级进程，高优先级进程将会霸占CPU。

展开

作者回复: 是的 铁汁

LDxy
2021-07-16

I/O 包这个概念第一次听说

展开

作者回复: 我搞 很多 新名词概念 哈哈

springXu
2021-07-16

因为内核的调度器也要被激活。如果没定时器中断，调度器的激活貌似只能靠其他中断，比如用户结束进程，或者进程抢占了其他被锁资源，才能激活进程调度器。

展开

作者回复: 正确 正确

1